Supposition Interface

March 7, 1985

1. Background

This is my response to RJ's WorkBench Specifiction, rough pass #1, dated 5 March 1984 (sic). I have tried to provide a set of calls that provide the functionality specified in that document, but many of the calls have been repackaged to make the interface easier to implement. I also tried to outline the interface at a deeper level, listing the arguments and return values of the various calls.

There are some issues in RJ's original document that are not directly implementable or have not yet been worked out, and need to be done before Supposition can be written. Those issues appear after the interface specification.

I am calling the interface between WorkBench and the DOS "Supposition". Supposition is divided into two parts: those things that WorkBench will call to give it the information that it requires (such as opening an object) and those things that Supposition will inform WorkBench about.

2. Definitions

2.1. Supposition

Supposition is the interface between WorkBench and the Disk based operating system (DOS).

2.2. Path Name

For this document a file name is a pair of two values: a Tripos lock and a path description relative to that lock. Throughout this document then the term *PathName* (or just ***path*** for short) it refers to one of these pairs. Absolute pathnames can be refered to by using a null lock and fully specifying the path name.

2.3. Disks, Drawers, Objects, and Directories

An Object corresponds to a WorkBench displayable entity. Objects may be disks, tools, projects, or drawers.

A Drawer is a WorkBench object that contains other WorkBench objects. A Drawer may not also be a tool or a project. Several operations (such as OpenDrawer) only make sense when applied to drawers.

Tools and projects are implemented via DOS directories, as are drawers. The difference is that tools and projects are "leaf nodes" in the WorkBench representation. both projects and tools may contain subdirectories, but these subdirectories are normally invisible to WorkBench.

A disk corresponds to a particular pieces of magnetic media (as opposed to a particular drive). How? Disks are unique in the universe -- any two disks can be told apart. WorkBench can tell when a disk is either inserted or removed. WorkBench also knows if anything is active on the disk, so it can tell if it should keep the disk around. A disk is active (iconically displayed) when either the disk is in the drive, or when something on the disk is active.

Currently I sometimes draw a distinction between specific types of objects (disks, drawers), and generic objects in some of the calls. This is mostly for convenience in specifing what sort of objects it makes sense to pass to that routine; I hope that I can use the same data structure to represent all objects.

3. WorkBench to Supposition Interface

This section refers to the direct requests that can be made to Supposition.

3.1. RunTool()

```
struct RunningTool *
RunTool( ToolPath, argcount, arglist )
struct Path *ToolPath;
int argcount;
struct Path *arglist[];
```

RunTool starts a tool up and running. It takes a (possibly null) list of projects and/or arguments to pass on to the tool. It returns a handle on the tool that can be used to communicate with the tool. In addition this handle is used to detect when the tool has terminated. If the call did not succeed, then a **NULL** will be returned.

3.2. StopTool()

```
StopTool( Tool )
struct RunningTool *Tool;
```

can you do this? Monitor to see if it goes away? What about tools that close of their own accord?

StopTool attempts to close an active instance of a tool (e.g. a tool started by *RunTool*). It will probably use a signal to inform the tool that it should clean up and go away. The tool is allowed to defer or ignore a request to close itself, or query the user to see if he realy wanted to close this tool. **WorkBench** will be informed of the actual close via *DeadTool*.

3.3. OpenDrawer()

```
OpenDrawer( Parent )
struct OpenObject *Parent;
```

Abstractly, *OpenDrawer* does all the research required to open a new drawer on the screen. It reads all the entries in a parent drawer. It will fill in the *od_Child* entry in the parent's *OpenObject* structure to point to the first child drawer read in. Each child drawer will in turn be linked together, and all these siblings will point back at the parent drawer. Each child drawer on this list will have the required information obtained from the drawer description file, such as gadget, type, etc.

object?

*** We need a list of this information ***

OK FINE

Unlike the **WorkBench** spec, this call will **NOT** recurse down the directory tree.

If there is an error during the directory search a **NULL** will be returned.

There are some aspects of reading a directory that I don't yet understand. This call may change some as I get the details that I need.

3.4. CloseDrawer()

```
CloseDrawer( Object )
struct OpenObject *Object;
```

I don't know.

This call frees the resources tied up by an *OpenDrawer*. It should be done when the drawer is closed or ~~when WorkBench is exiting~~.

If there are child drawers currently open for this drawer the children will be updated to know that their parent is gone.

I don't know.

The drawer will have its usage count decremented. Any drawer that has a usage count of zero will have its memory freed.

3.5. OpenDisk()

```
struct OpenObject *
OpenDisk( Disk )
struct OpenDisk *Disk;
```

OpenDisk returns a handle on the root drawer of the specified disk. This handle can in turn be used to read in the contents of that drawer.

If the disk cannot be opened then a **NULL** is returned.

Why might a disk not open? Disk not initialized

3.6. CloseDisk()

CloseDisk(Disk)
struct OpenDisk *Disk;

The inverse operation of *OpenDisk* is *CloseDisk*. It should be called when the disk is inactive. A disk should be marked ~~inactive when all drawers, tools, and~~ projects from that disk have been closed. This means that we will have to keep a usage count for items on that disk.

exactly.

3.7. ObjectInfo()

SetObjectInfo(Object, Info)
GetObjectInfo(Object, Info)
struct OpenObject *Object;
struct ObjectInfo *Info;

Conventionally there is a set of information that is kept about an object. While we do not know the complete set of this information, it includes a description of its gadget, its type (tool, project, drawer, etc), some description description text, and information that is specific to the object type. In addition this file may provide handles to implement WorkBench operations, such as copying or removing the object.

GetObjectInfo obtains this information from the disk. *SetObjectInfo* writes the information back out to disk.

3.8. Object Manipulation

MoveObject(Object, NewPath)
CopyObject(Object, NewPath)
Struct OpenObject *Object;
Struct PathName *NewPath;

These calls manipulate objects on the disk. The implementation will be object dependent, and may be implemented by the object itself (instead of by Supposition).

every move/copy of a non-disk object is move/copy directory.

4. Supposition to WorkBench Interface

These calls are the ones that supposition will call into WorkBench.

4.1. Disk Changes

DiskInserted(Disk)
DiskRemoved(Disk)
struct OpenDisk *Disk;

These WorkBench entry points will be called whenever a disk is inserted or removed. They are passes an identifier for the disk that is inserted.

4.2. Object Changes

ObjectChanged(Object)
struct OpenObject *Object;

ObjectChanged is called when there the status of an object has been detected. This change may or may not actually require any WorkBench display changes; instead it says that changes **could** be required. The WorkBench data strucures are not actually maniputed; instead it is just marked as needing attention.

If we can develop a locking mechanism for the object data structures then this call will not be needed; instead the objects may be updated directly and some sort of WorkBench refresh routine called if and only if there is a displayable change.

4.3. DeadTool()

```
DeadTool( Tool )
struct RunningTool *Tool;
```

def StopTool() for notes

DeadTool informs **WorkBench** that a previously started tool has terminated.

5. Open Issues

5.1. Memory Allocation

Lots of open question

We will need lots of dynamic data to implement **WorkBench**. Where we get this data from, who cleans it up, and when it gets cleaned up are all open issues.

I tried to outline some ideas of when things should be freed in some of the calls.

5.2. Control Flow

We will need to structure the program so that both **WorkBench** and **Supposition** can get their respective input. **WorkBench** needs to recieve input events from intuition; **Supposition** needs to recieve notifiction from the **DOS**, a timer, and processes that have terminated. In addition some events will happen asynchronously; the kindest description I can find for how Tripos deals with asynchrous events is "obsure".

This is why it should be done program, not here.

I spoke with Tim King and we decided that the method that makes the most sense is to structure **WorkBench** the same as Tripos utilities that need to be asynchronous: they are structured as coroutines. This implies some methods on how **Supposition** and **WorkBench** should work. I am currently working on the low level assembly code to give C programs access to BCPL style coroutines.

I DON'T THINK I WANT TO SPEND TIME TO LEARN DOING IT THIS WAY. JUST SET UP RAW INPUT EVENTS.

5.3. Shared Data Structures

There will be many shared data objects between **Supposition** and **WorkBench**. We will need to develop a locking protocol so that these objects are always consistent.

5.4. DOS information

Currently plans have the **DOS** keeping information on whether an object with a lock has been changed. **Supposition** can poll this information periodically to see if any of the objects that it cares about have changed. I believe that will will allow us to naturally check to see if a folder has had new items installed in it or removed.

A more difficult problem is tracking the open status of projects. It has been suggested that **WorkBench** should reflect the current state of objects; if a project is opened via a non-**WorkBench** path then the icon should reflect this. One problem is that there is currently no way that I know of to do this. Another is that icons will be flickering on and off without the user interacting with **WorkBench** -- this will be confusing. Finally we don't have a **DOS** level definition of what it means to "open a project". We know how to pass a project on to a tool; however what that tool does with that project is completely up to it. What if the tool starts accessing another project? **WorkBench** has no knowledge of the semantics of the interactions between tools and projects.

One compromise that would be simple to implement is to allow a tool to change the state of a project. We could provide a call to manipulate projects through their three states (closed, currently being read, or currently being written).

OPEN/CLOSE "FLICKERING" WILL BE INFORMATIVE AND, LIKE PANEL LIGHTS FLASHING, EVEN AMUSING, AND WILL NOT HAPPEN WITH TMP FILES

5.5. Process relationships

To be filled in

5.6. Locating and Naming Tools

To be filled in

5.7. Rodent Removal

Many people have a large resistance to mice and feel that they are not a convenient way to specify objects. We previously decided that we wanted to have an additional method of specifying objects and actions on them. This alternate method would still be graphical in nature: objects would highlight, move, and open the way they do with a mouse. However the method of specifing objects would be different: via function keys or by typing in names and control sequences.

I think that there is a natural way for this to happen. However, this way would require **WorkBench** to be sigificantly restructured. We need to think about how important this is and whether we can do it later.

There's an entire non-icon WBench in my head.

brought to you by
andy finkel